Computer Science 210

Computer Systems 1

Lecture 11

The Instruction Cycle
Ch. 5: The LC-3 ISA

Credits: "McGraw-Hill" slides prepared by Gregory T. Byrd, North Carolina State University

---

Instruction Processing: FETCH

Finite State Machine

Load next instruction
(at address stored in PC) from memory
into Instruction Register (IR)

- Copy contents of PC into MAR
- Send "read" signal to memory
- Copy contents of MDR into IR

Then increment PC, so that it points to the next instruction in sequence.

- PC becomes PC+1.

Instruction Processing: DECODE

First identify the opcode.

- In LC-3, this is always the first four bits of instruction
- A 4-to-16 decoder asserts a control line corresponding to the desired opcode

Depending on opcode, identify other operands from the remaining bits

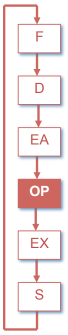
- Example:
  - for LDR, last six bits is offset
  - for ADD, last three bits is source operand #2

Instruction Processing: EVALUATE ADDRESS

For instructions that require memory access, compute address used for access

Examples:
- add offset to base register (as in LDR)
- add offset to PC
- add offset to zero

Instruction Processing: FETCH OPERANDS



Obtain source operands needed to perform operation.

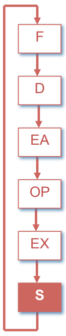
Examples:
- load data from memory (LDR)
- read data from register file (ADD)

Instruction Processing: EXECUTE

Perform the operation, using the source operands.

Examples:
- send operands to ALU and assert ADD signal
- do nothing (e.g., for loads and stores)

Instruction Processing: STORE RESULT



- Write results to destination (register or memory)

- Examples:
  - result of ADD is placed in destination register
  - result of memory load is placed in destination register
  - for store instruction, data is stored to memory
    - write address to MAR, data to MDR
    - assert WRITE signal to memory

Changing the Sequence of Instructions

In the FETCH phase,
we increment the Program Counter by 1

What if we don't want to always execute the instruction that follows this one?

- examples: loop, if-then, function call

Need special instructions that change the contents of the PC.

These are called ***control instructions***

- **jumps** are unconditional – they always change the PC
- **branches** are conditional – they change the PC only if some condition is true (e.g., the result of an ADD is zero)

Example: LC-3 JMP Instruction

Set the PC to the value contained in a register. This becomes the address of the next instruction to fetch.

				unused								unused			
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JMP				0	0	0	Base			0	0	0	0	0	0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	1	0	0	0	0	0	0

"Load the contents of R3 into the PC."

Instruction Processing Summary

Instructions look just like data – it's all interpretation.

Three basic kinds of instructions:
- computational instructions (ADD, AND, …)
- data movement instructions (LD, ST, …)
- control instructions (JMP, BR, …)

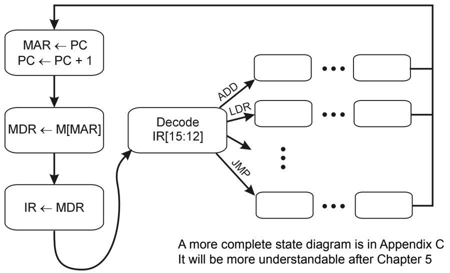
Six basic phases of instruction processing:

F → D → EA → OP → EX → S

- not all phases are needed by every instruction
- phases may take variable number of machine cycles

Control Unit State Diagram

The control unit is a state machine. Here is part of a simplified state diagram for the LC-3:

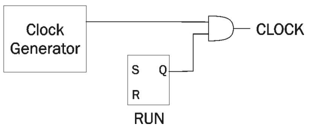


A more complete state diagram is in Appendix C
It will be more understandable after Chapter 5

Stopping the Clock

Control unit will repeat instruction processing sequence as long as clock is running
- If not processing instructions from your application, then it is processing instructions from the Operating System (OS)
- The OS is a special program that manages processor and other resources

To stop the computer:
- AND the clock generator signal with ZERO
- When control unit stops seeing the CLOCK signal, it stops processing



Instruction Set Architecture

ISA = All of the ***programmer-visible*** components and operations of the computer

- **memory organization**
  - address space -- how may locations can be addressed?
  - addressibility -- how many bits per location?
- **register set**
  - how many? what size? how are they used?
- **instruction set**
  - opcodes
  - data types
  - addressing modes

ISA provides all information needed for someone that wants to write a program in **machine language**
(or translate from a high-level language to machine language)

LC-3 Overview: Memory and Registers

Memory

- address space: 2^{16} locations (16-bit addresses)
- 65,536 memory address
- addressability: 16 bits

Registers

- temporary storage, accessed in a single machine cycle
  - accessing memory generally takes longer than a single cycle
- eight general-purpose registers: R0 - R7
  - each 16 bits wide
  - 4 bits to uniquely identify a register?
- other registers
  - not directly addressable, but used by (and affected by) instructions
  - PC (program counter), condition codes

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD^{+}	0001				DR			SR1			0	00		SR2		
ADD^{+}	0001				DR			SR1			1	imm5				
AND^{+}	0101				DR			SR1			0	00		SR2		
AND^{+}	0101				DR			SR1			1	imm5				
BR	0000				n	z	p	PCoffset9								
JMP	1100				000			BaseR			000000					
JSR	0100				1	PCoffset11										
JSRR	0100				0	00		BaseR			000000					
LD^{+}	0010				DR			PCoffset9								
LDI^{+}	1010				DR			PCoffset9								
LDR^{+}	0110				DR			BaseR			offset6					
LEA^{+}	1110				DR			PCoffset9								
NOT^{+}	1001				DR			SR			111111					
RET	1100				000			111			000000					
RTI	1000				000000000000											
ST	0011				SR			PCoffset9								
STI	1011				SR			PCoffset9								
STR	0111				SR			BaseR			offset6					
TRAP	1111				0000				trapvect8							
reserved	1101															

LC-3 Overview: Instruction Set

Opcodes
- 15 opcodes
- ***Operate*** instructions: ADD, AND, NOT
- ***Data movement*** instructions: LD, LDI, LDR, LEA, ST, STR, STI
- ***Control*** instructions: BR, JSR/JSRR, JMP, RTI, TRAP
- some opcodes set/clear *condition codes*, based on result:
  - N = negative, Z = zero, P = positive (> 0)

Data Types
- 16-bit 2's complement integer

Addressing Modes
- How is the location of an operand specified?
- non-memory addresses: *immediate*, *register (direct)*
- memory addresses: *PC-relative*, *indirect*, *base + offset*

Operate Instructions

Only three operations: **ADD, AND, NOT**

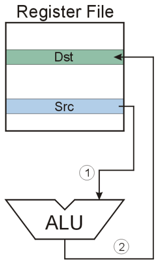
Source and destination operands are **registers**
- These instructions ***do not*** reference memory.
- ADD and AND can use "immediate" mode, where one operand is hard-wired into the instruction

Will show dataflow diagram with each instruction
- illustrates ***when*** and ***where*** data moves to accomplish the desired operation
- Watch the video http://youtu.be/yZChqRqPluI

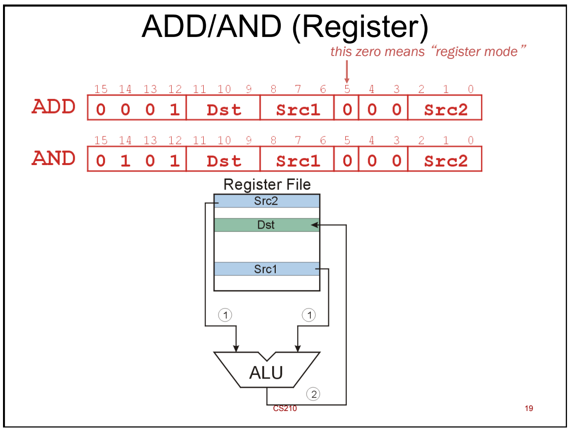
NOT (Register)

unused bits

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
NOT	1	0	0	1	Dst			Src			1	1	1	1	1	1



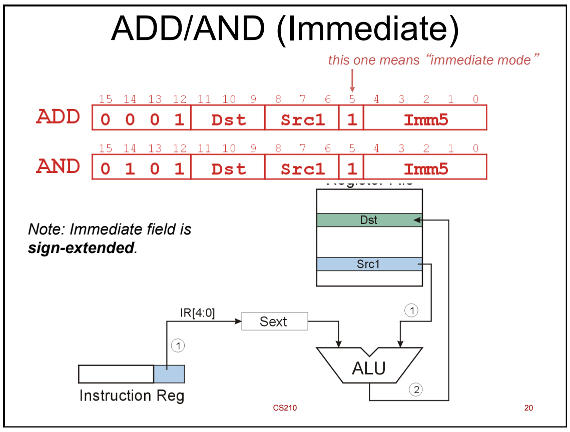
Note: Src (source) & Dst (destination) could be the *<u>same</u>* *register.*



ADD/AND (Register)

this zero means "register mode"

	15	14	13	12	11 10 9	8 7 6	5	4	3	2 1 0
ADD	0	0	0	1	Dst	Src1	0	0	0	Src2
AND	0	1	0	1	Dst	Src1	0	0	0	Src2

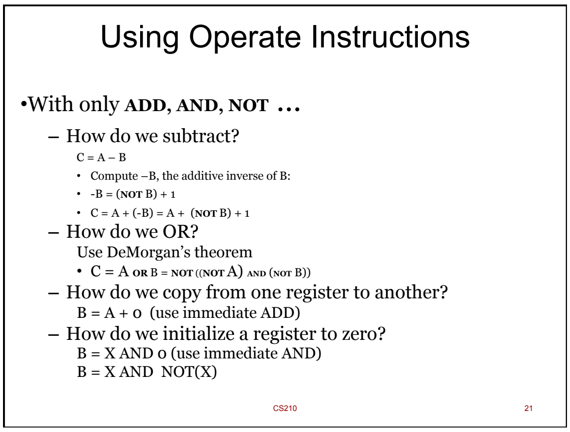


ADD/AND (Immediate)

this one means "immediate mode"

	15	14	13	12	11 10 9	8 7 6	5	4 3 2 1 0
ADD	0	0	0	1	Dst	Src1	1	Imm5
AND	0	1	0	1	Dst	Src1	1	Imm5

Note: Immediate field is ***sign-extended****.*



Using Operate Instructions

- With only **ADD, AND, NOT** …
  - How do we subtract?
    - C = A – B
    - Compute –B, the additive inverse of B:
    - -B = (NOT B) + 1
    - C = A + (-B) = A + (NOT B) + 1
  - How do we OR?
    - Use DeMorgan's theorem
    - C = A OR B = NOT ((NOT A) AND (NOT B))
  - How do we copy from one register to another?
    - B = A + 0 (use immediate ADD)
  - How do we initialize a register to zero?
    - B = X AND 0 (use immediate AND)
    - B = X AND NOT(X)

PC-Relative Addressing Mode

Want to specify address directly in the instruction
- LC-3 memory has 2^{16} (65,536) memory addresses
- Each address is 16 bits, and so is as long as an instruction!
- After subtracting 4 bits for opcode and 3 bits for register, we have *9 bits* available for address
- Only 512 addresses are reachable with 9 bits

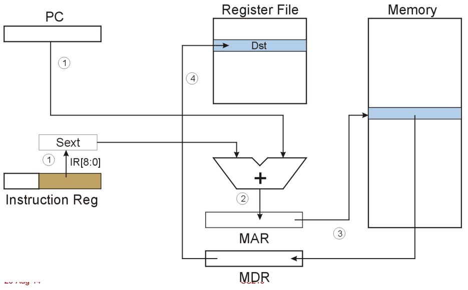
Solution:
- Use the 9 bits as a ***signed offset*** from the current PC address.

9 bits: $-256 \le \text{offset} \le +255$
Can form any address X, such that:
$PC - 256 \le X \le PC + 255$

Effectively we can access any part of the memory ~256 bits around the 16 bit address in the PC
Since we can change the address in the PC we can access the entire memory

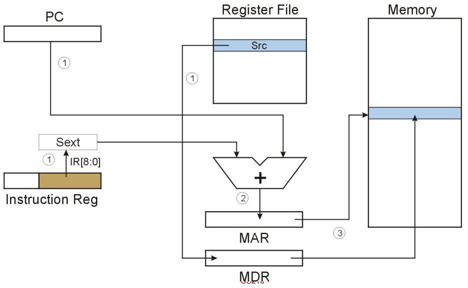
LD (PC-Relative)

	15	14	13	12	11 10 9	8 7 6 5 4 3 2 1 0
LD	0	0	1	0	Dst	PCoffset9



ST (PC-Relative)

	15	14	13	12	11 10 9	8 7 6 5 4 3 2 1 0
ST	0	0	1	1	Src	PCoffset9



Indirect Addressing Mode

With PC-relative mode, can only address data within 256 words of the instruction.

- What about the rest of memory?
- What if we don't want to change the value in the PC

Solution #1:

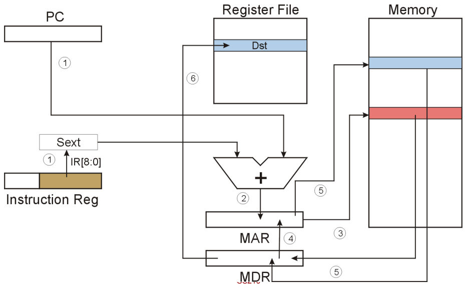
- Read address from memory location, then load/store to that address.

First address is generated from PC and IR (just like PC-relative addressing), then content of that address is used as target for load/store.

Watch video: http://youtu.be/cDaPPXyYbH0

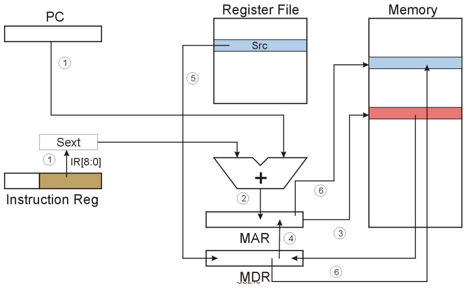
LDI (Indirect)

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LDI	1	0	1	0	Dst			PCoffset9								



STI (Indirect)

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
STI	1	0	1	1	Src			PCoffset9								



Base + Offset Addressing Mode

With PC-relative mode, can only address data within 256 words of the instruction.

- What about the rest of memory?

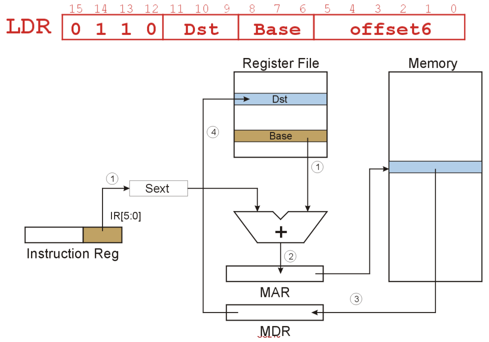
Solution #2:

- Use a register to generate a full 16-bit address

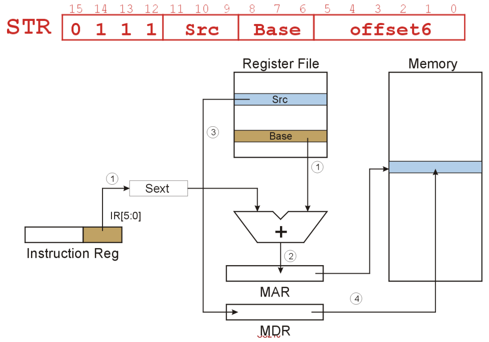
4 bits for opcode, 3 for src/dest register, 3 bits for ***base*** register -- remaining 6 bits are used as a ***signed offset***

- **Offset is *sign-extended* before adding to base register**

LDR (Base+Offset)



STR (Base+Offset)

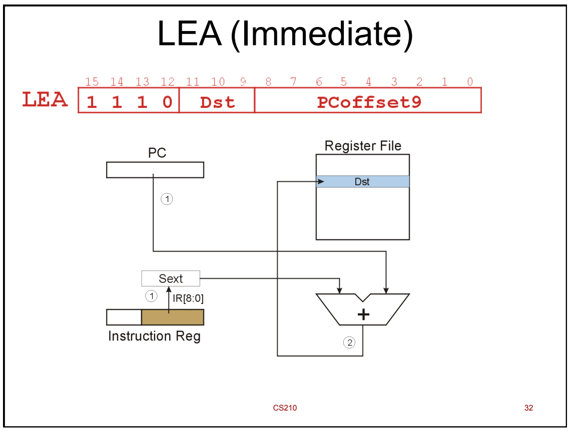


Load Effective Address

Computes address like PC-relative (PC plus signed offset) and stores the result into a register.

Note: The *address* is stored in the register, not the contents of the memory location

LEA (Immediate)



Control Instructions

Used to alter the sequence of instructions (by changing the Program Counter)

Conditional Branch
- branch is *taken* if a specified condition is true
  - signed offset is added to PC to yield new PC
- else, the branch is *not taken*
  - PC is not changed, points to the next sequential instruction

Unconditional Branch (or Jump)
- always changes the PC
- watch the video http://youtu.be/GF1z7MEa-pk

TRAP
- changes PC to the address of an OS "service routine"
- routine will return control to the next instruction (after TRAP)

Condition Codes

LC-3 has three **condition code** bits:
- **N** -- negative
- **Z** -- zero
- **P** -- positive (greater than zero)

Set by any instruction that writes a value to a register
(ADD, AND, NOT, LD, LDR, LDI, LEA)

Exactly *one* will be set at all times
- Based on the last instruction that altered a register

Branch Instruction

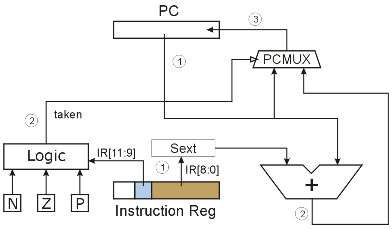
Branch specifies one or more condition codes.
If the set bit is specified, the branch is taken.

- PC-relative addressing:
  target address is made by adding signed offset (IR[8:0]) to current PC.
- Note: PC has already been incremented by FETCH stage.
- Note: Target must be within 256 words of BR instruction.

If the branch is not taken,
the next sequential instruction is executed.

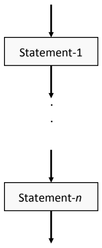
BR (PC-Relative)

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BR	0	0	0	0	n	z	p	PCoffset9								



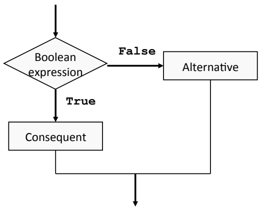
Control Structures: Sequence

```
begin
statement-1
.
.
statement-n
end
```



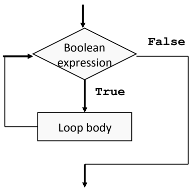
Control Structures: Selection

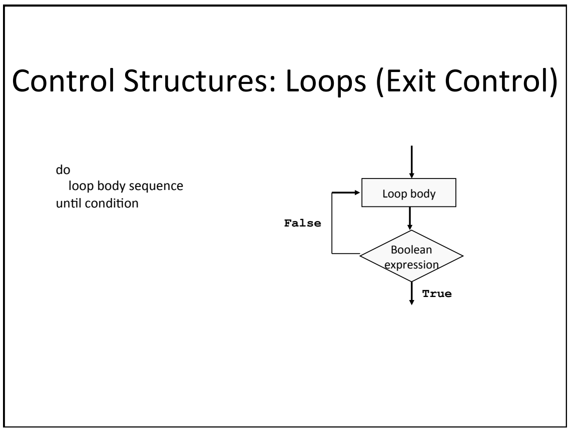
```
if condition then
   consequent sequence
else
   alternative sequence
```



Control Structures: Loops (Entry Control)

```
while condition do
   loop body sequence
```





Control Structures: Loops (Exit Control)

```
do
  loop body sequence
until condition
```

LC-3 Control Instructions

- Conditional branch (BR)
- Absolute branch (JMP)
- Procedure call (JSR, JSRR, RET, RTI)
- System call (TRAP)

Condition Codes

- 3 single-bit registers named N, Z, and P
- Exactly one will be set at all times

0	1	0
N	Z	P

- Automatically set by any instructions that writes data to a register (ADD, AND, NOT, LD, LDR, LDI, LEA)

Example: Subtract 1 from R3

`add R3, R3, -1`

Circuitry sets condition codes after **`add`** executes

0	1	0
N	Z	P

When R3 = 0

0	0	1
N	Z	P

When R3 > 0

1	0	0
N	Z	P

When R3 < 0

Conditional Branch (BR)

- Alters a sequence of instructions by changing the PC
- Branch is taken if the condition is true
- Signed offset is added to PC if condition is true; otherwise, PC not changed

---

Conditional Branch

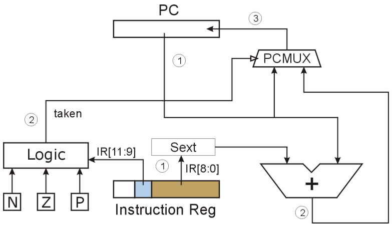
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	N	Z	P	PC offset								

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	1	1	0	1	1	0	0	1
BR				N	Z	P	x0D9								

Offset is sign-extended and added to the incremented PC

Destination must be no more than +256 or -255 from the BR itself

Data Path for BR



Example: An Unconditional Branch

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	N	Z	P	PC offset								

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	0	1	1	0	1	1	0	0	1
BR				N	Z	P	x0D9								

At least one condition code is guaranteed to match the codes in this instruction

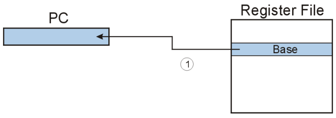
Jump Instruction (JMP)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	Base			0	0	0	0	0	0	0	0	0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0

Contents of the base register are copied to the PC

Can go anywhere in memory!



Trap Instruction (TRAP)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	0	**Trap vector (8 bits)**							

OS service routine

Operation coded in trap vector

R0 used for input and output

After completion, PC is set to instruction following the TRAP

Trap Instruction (TRAP)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	0	**Trap vector (8 bits)**							

x20 (GETC) - waits for the keyboard interrupt and reads a single character and converts the key value into an ASCII character. The character is not echoed to the console screen, it is simply read and stored into a register

x21 (OUT) - writes the character currently in R0 onto the console display

X22 (PUTS) - writes an array of characters or string to the console (the data is converted into ASCII before printing to the screen). The first character is stored in R0 continues down the array until until the program finds data reading 0x0000

X23 (IN) - waits for character input, the character is echoed back to the screen and is also stored into R0 as an ASCII value

Trap Instruction (TRAP)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	0	**Trap vector (8 bits)**							

X24 (PUTSP) - recording input strings, each register will hold a pair of characters and the address of the first character is stored in R0. The user writes into the console and the program stores the characters into an array. Writing terminates with the occurrence of 0x0000

x25 (HALT) - used for ending programs, it doesn't terminate the program, it simply stops execution by the use of a forever loop